[redacted]

EXHIBIT G

SCOUT: A PATH-BASED OPERATING SYSTEM

by

David Mosberger

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

1 9 9 7

by all other domains. A third would involve accessing the path object through a system call-like interface. Most likely, an actual implementation would use a combination of the three proposed solutions, but the key point is that there do not seem to be any unusual difficulties in defining paths that cross multiple protection domains.

Vertical partitioning does not appear to be problematic either. The entire path would be contained in its own protection domain and the domain would have to be crossed only when moving from a module into a path or vice versa. Ideally, paths directly connect modules representing device pairs, so the number of domain crossings for moving data from a source device to sink device would be two, just as is the case with a traditional, monolithic kernel based system [95].

3.4 Demultiplexing

So far, we have not discussed the issue of how the appropriate path is found for a given message. In many cases, this is trivial. For example, a path is often used like a file descriptor, a window handle, or a socket descriptor. In these cases, paths are created specifically for communicating a certain kind of data and the appropriate path is known from the context in which it is being used. In other cases, however, there is no such context and the appropriate path is implicitly determined by the data itself. The most notable area where this is the case is for the networking subsystem. Networking protocols typically require hierarchical demultiplexing for arriving network packets. For example, when a network packet arrives at an Ethernet adapter [65], the Ethernet type field needs to be looked up to determine whether the packet happens to be, e.g., an IP packet or an ARP packet. If it is an IP packet, it is necessary to look up the protocol field to determine whether it is a UDP or TCP packet, and so on. This hierarchical demultiplexing is suboptimal since as more knowledge becomes available with each demultiplexing step, another path might become more suitable to process that packet.

3.4.1 Scout Packet Classifier

To alleviate the hierarchical demultiplexing problem, Scout uses a packet classifier that factors all demultiplexing operations to the earliest possible point. This lets Scout pick a is interesting. Consider Figure 3.6. It would be preferable to process every incoming packet destined for UDP using path p1. Unfortunately, this ideal case cannot always be achieved. For example, the UDP packets may have been fragmented by IP, or IP might have employed a non-trivial data transformation, such as encryption or compression. In all three cases, some protocol headers may not be available to the classifier. Scout copes with this problem by employing short paths as necessary. In the example, an IP fragment would have to be processed first by path p2 and then by path p4, even though this would be less optimal than processing with p1.

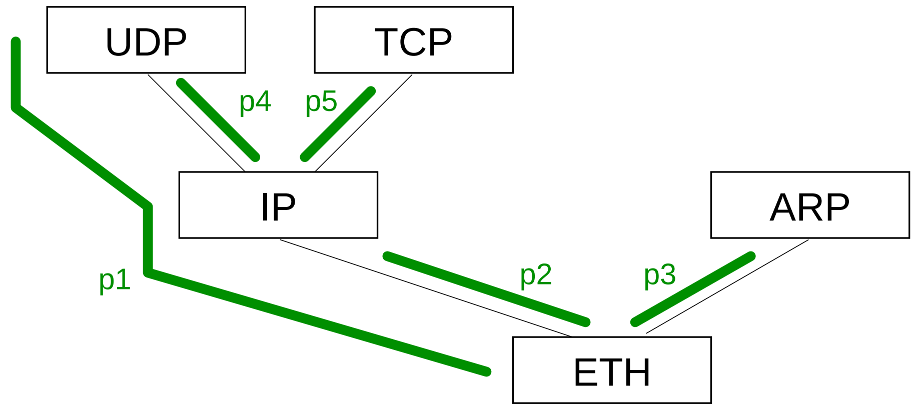

Figure 3.6: Paths Versus Classifiers

The specifics of how the partial classification problem is solved in Scout are explained next. As previously suggested, the Ethernet module (ETH) employs a classifier to decide whether a packet should be processed using path p1, p2, or p3. Suppose an IP fragment that does not contain the higher-level headers arrives at the Ethernet adapter. Without higher-level headers, the best the classifier may be able to do is to determine that path p2 is appropriate for processing the fragment.[1] Once IP receives the fragment through p2, it will buffer the fragment until the entire datagram has been reassembled. At that point, IP needs to make a dynamic routing decision to find out where to send the complete datagram

[1]Note that, strictly speaking, since the UDP protocol identifier is stored in the IP header, in this particular example it would be possible to classify the packet to path p1, but this is only because the example is artificially simple. In a more realistic scenario, path p1 would extend beyond UDP, meaning that the fragment could not be classified to p1.

next. IP can do this by running its *own* classifier on the complete datagram, which, with a UDP packet, will tell IP that p4 should be used.

This example shows that, in Scout, a classifier is not something specific to network drivers, but instead is a mechanism that can be employed by any module that needs to make a dynamic routing decision based on the contents of the data being communicated. Typically, each network device driver uses a classifier which is invoked when a packet receive interrupt is processed, but other modules may use their own classifiers if necessary.

3.4.3 Realizing the Scout Classifier

Since Scout is a modular system, we would like to be able to express classification in a modular fashion as well. That way, the complete classifier can be built automatically from partial classification algorithms that are specified by the module that appear along a path. Note that a modular specification of the classifier does not necessarily imply a modular implementation, though this is true for the current Scout implementation.

As discussed earlier, classifiers are used in Scout by any module that may need to make a dynamic routing decision based on the contents of a message. The task of a classifier can therefore be described as: given a module *m* and a message, determine a path that is appropriate for processing the given message. Since it is module *m* that is making the routing decision, the path must start at that module.

The classification task can be implemented in an iterative fashion using partial classifiers of the following form: given a set of paths and a message, determine the subset of paths that qualify for the processing of the given message, the module that needs to make the next (refining) classification (if any), and the message for that next module. The message for the next module is normally the same as the original message, except that the protocol header at the front of the message will have been stripped off. This scheme works as long as the classification problem is locally hierarchical. The ramifications of this constraint will be explained in detail in Section 3.4.3.1. For now, we assume that the Scout classification problem satisfies this constraint. Using such partial classifiers, the problem can be solved with the pseudo-code shown in Figure 3.7. Lines 2–5 of the pseudo-code determine the set of paths that either begin or end at module `m`. This path set